CMR Bulk Update

Current Assumptions

- Starting off only updating certain fields - GCMD keyword fields: Science Keywords, Location Keywords, Data Centers, Platforms Instruments
- Cancel not part of initial implementation
- If multiple bulk update processes running updating the same field on the same collection, no guarantee that you'll get the latest update
- When bulk updating a collection, the collection is saved in umm-json regardless of the original native format

Tables

2 tables for tracking the status of a bulk update task:

Bulk update task table - overall status

- Task Id
- Provider Id
- Request JSON body
- Status
- Status message

Bulk update status table - individual collection statuses

- Task Id
- Concept Id
- Status
- Status message

API Endpoints

Ingest Endpoint

POST ingest/providers/<provider-id>/bulk-update/collections

POST JSON:

- List of collections to update - either concept id or short-name/version *(short-name/version currently not supported)*
- Field to update - required
- Type of update to make - required
- Update value
- Find value

Field to update and type of update are enumerations.

The update and find values are the UMM object or a subset of the UMM object. For example, the science keywords update value would be the full science keyword {"Category": "Cat1"...} and the find value would take all or part of a science keyword and match on the given fields.

The ingest endpoint:

- Validates the parameters against a JSON schema and rules i.e. the new value and find value can be required based on type of update.
- Checks ACL update permissions for provider
- If short-name/version supplied, get concept ids *(on hold for now - will be later functionality)*
- Writes to the task status and collection status tables in a transaction and gets the task id
- Queues a bulk update message
- Returns status code and task Id if no error, otherwise error messages

Validations

Bulk update POST body validations:

- Short-name/version Ids required or concept-ids required, min 1 item in the list
- Type of change required
- Field to update required
- Field to update valid (in map of fields)
- If NOT type FIND_AND_REMOVE, New value required
- If type FIND_AND_REMOVE, FIND_AND_REPLACE or FIND_AND_UPDATE, Find value required

Sample POST JSON

```
{
    "concept-ids": [
        "C1",
        "C2",
        "C3"
    ],
    "update-type": "FIND_AND_UPDATE",
    "update-field": "SCIENCE_KEYWORDS",
    "find-value": {
        "Category": "EARTH SCIENCE",
        "Topic": "HUMAN DIMENSIONS"
    },
    "update-value": {
        "Category": "EARTH SCIENCE",
        "Topic": "HUMAN DIMENSIONS",
        "Term": "ENVIRONMENTAL IMPACTS",
        "VariableLevel1": "HEAVY METALS CONCENTRATION"
    }
}
```

Status Endpoints

Both require ACL read permissions for provider.

Endpoint to get all task statuses by provider: GET ingest/providers/<provider-id>/bulk-update/collections/status

Returns list of:

- Task Id
- Task Status enum ("IN_PROGRESS", "COMPLETE")

For each task for that provider.

Endpoint to get individual task status: GET ingest/providers/<provider-id>/bulk-update/collections/status/<task-id>

Returns:

- Task Status
- Task Status Message
- For each failed collection:
    - Concept Id
    - Collection update status message

Bulk Update Processing

For the update value, a full or partial UMM object is sent up. For example, a science keyword can be sent up with one or more fields. If a partial science keyword is sent up and the update type is find and update, the partial value would be merged with the original value. For example, if the original UMM has a science keyword of EARTH SCIENCE > HUMAN DIMENSION > ENVIRONMENTAL IMPACTS and we send up an update value of "Topic": "HUMAN DIMENSIONS", only the topic will be updated. They category and term will stay the same and the resulting science keyword will be EARTH SCIENCE > HUMAN DIMENSIONS > ENVIRONMENTAL IMPACTS.

Platforms can have nested instruments. Currently when doing a bulk update operation on instruments it will be applied to instruments in each platform. Add to existing will add the instrument to each platform's list of instruments. Find and replace, find and remove, and find and update will go through each platform's list of instruments. Clear all and replace will work similarly.

Bulk Update Queue

Processing a bulk update message:

- For each collection, publish individual bulk update messages

Processing a message on the queue:

- Check if the collection status is cancelled. If so, skip. *(This will not be implemented in the first iteration)*
- Pull the collection from metadata DB
- Translate collection to UMM (or latest version of UMM, if already in a previous version of UMM)
- Make field update change
- Perform ingest validations: i.e. concept, schema, business rules, warnings
- Save to metadata db with revision Id
    - **Save the collection with current version of umm-json as native format**
    - If concurrency failure, fail the message and will be retried
- In a transaction
    - Update the bulk update status table for the collection
    - Check to see if the overall bulk update is done and update the bulk update task table
- If bulk update task is complete, re-index provider collections

Future Work

AWS

Use the above design with an SNS/SQS message queue. For each collection, we write a message to the queue to process that collection (may want to do this in batches).

We have a lambda function that does the collection message processing outlined above. The lambda function would live in a separate project that refers to umm-spec-lib to do the updates. Use transmit lib to talk to metadata db.

Considerations:

- Lambda start-up time
- Lambda limitations in terms of memory, disk space, and execution time.
- Running too many concurrent lambdas

Testing

- Unit test update function in umm-spec-lib
- Test to make sure states are tracked and updated correctly
- Be able to run everything locally/in-memory

Cancel Operation

Add an endpoint to cancel a bulk update by task id. Need to check ACL permissions. Mark all collection entries in the bulk update status table as cancelled if they have not already been processed. Update the bulk update task table with the overall status: cancelled or partial cancel.

DB Cleanup

Add a process that goes through and cleans up old bulk update db status rows periodically.

Performance Testing Results

Tested bulk updates in workload on June 5, during an ingest run. Tested with varying number of concept-ids, from 100 to 2000, and different combinations of bulk update types.

Number of collections	Average time to completion (s)	Longest run (s)	Notes
100	16.1	42.0	
500	30.2	31.2	
500, multiple bulk updates, different collections	50.9	51.0	Running 2+ bulk updates at the same time, different concept ids
500, multiple bulk updates, same collections	221.5	336.9	Running 2+ bulk updates at the same time, same concept ids
1000	63.5	63.8	
2000	168.3	169.0	

Findings Summary

- There was no significant negative impact to average ingest rate during the bulk updates. There were slight spikes during bulk updates, but the increase did not necessarily correlate to the number of collections being updated or the number of concurrent bulk updates running. The spike were always less than 0.1s.
- Running multiple bulk updates at the same time, even on different collections, impacts the completion time of all bulk updates.
- Running multiple bulk updates at the same time on the same collections more than doubles the completion time. This is because concurrency conflicts will result in re-queueing and re-processing the collection, resulting in the overall bulk updating taking longer.

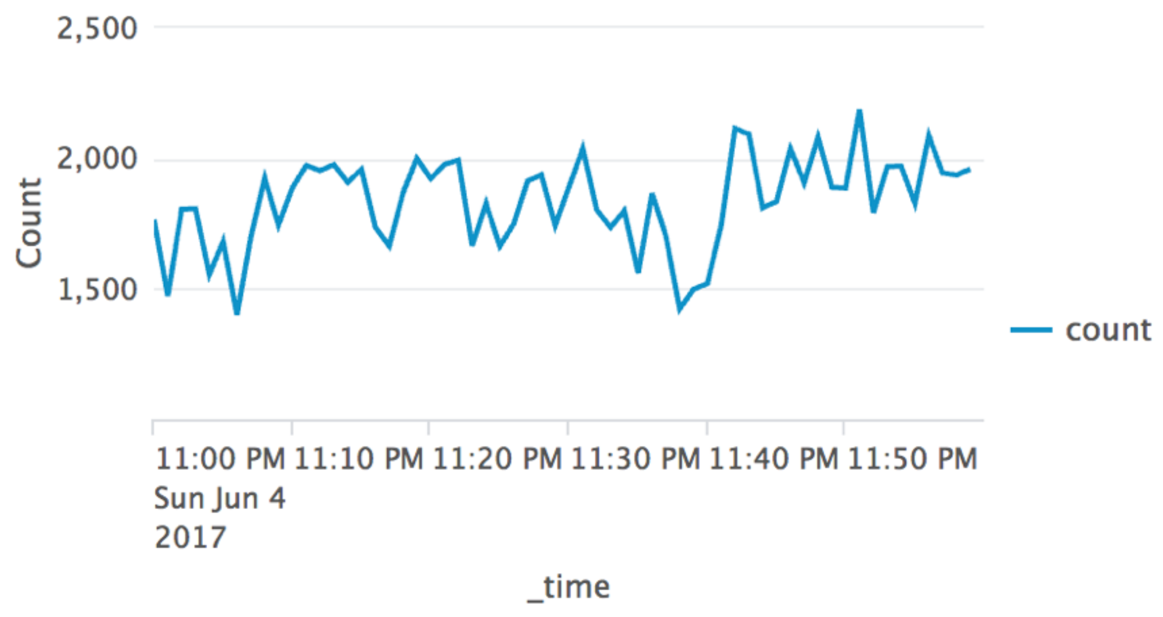
Graphs

The graphs below show two instances of bulk update during workload testing.

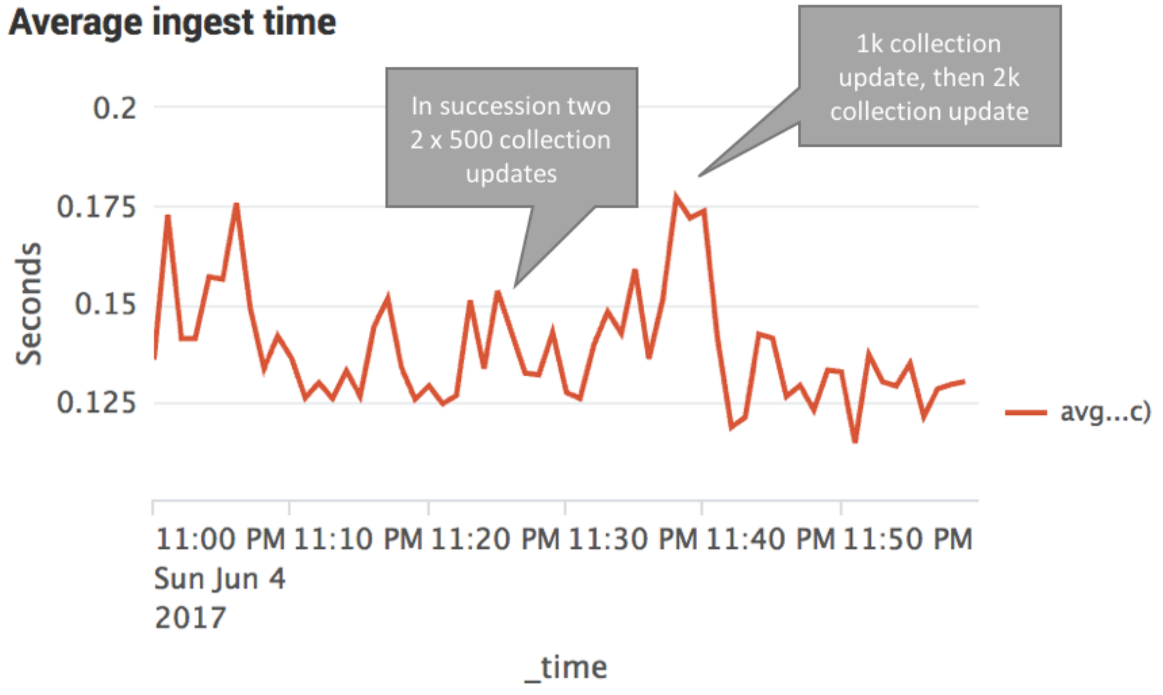
In the first graph, bulk update testing was performed consistently with numerous bulk updates occurring during the period of 11:10 to 11:40. The graph highlights the instance of simultaneous updates around 11:25, but otherwise bulk updates were happening in succession. The bulk updates ranged from 100 collections to 2000, with each scenario being tested multiple times.

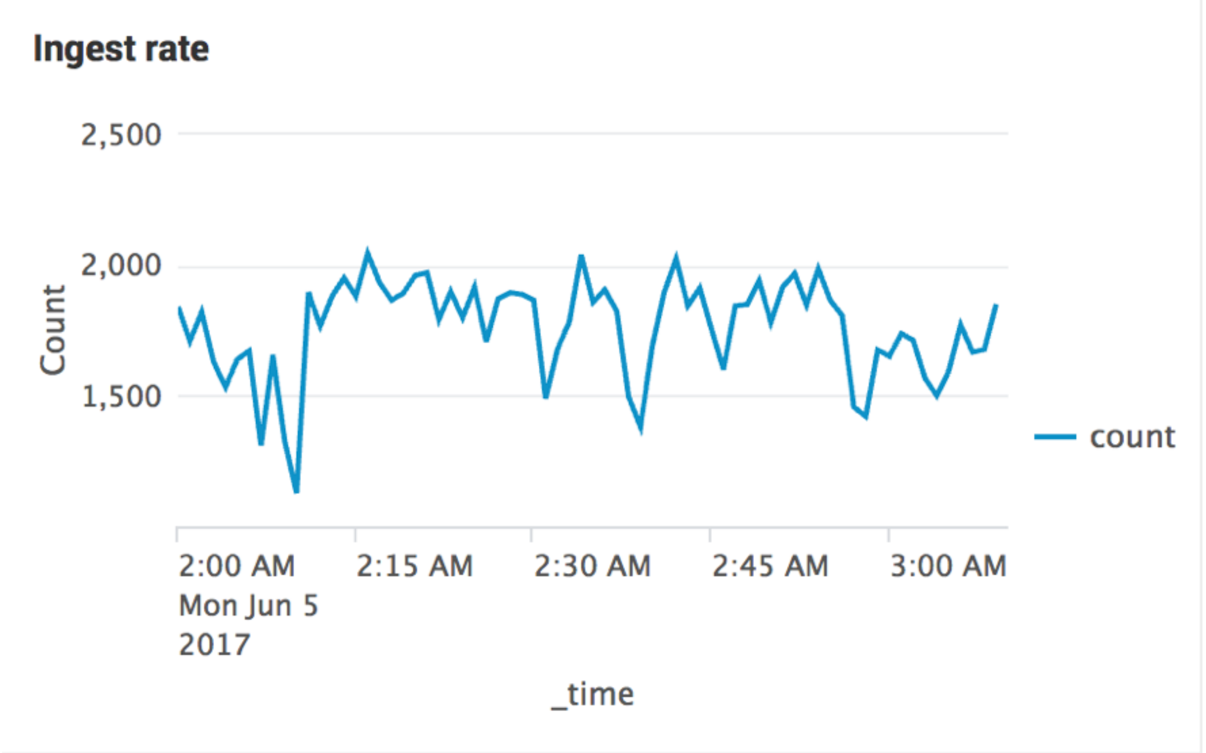
The second graph shows the impact of multiple simultaneous bulk updates of 500 collections occurring throughout the hour.

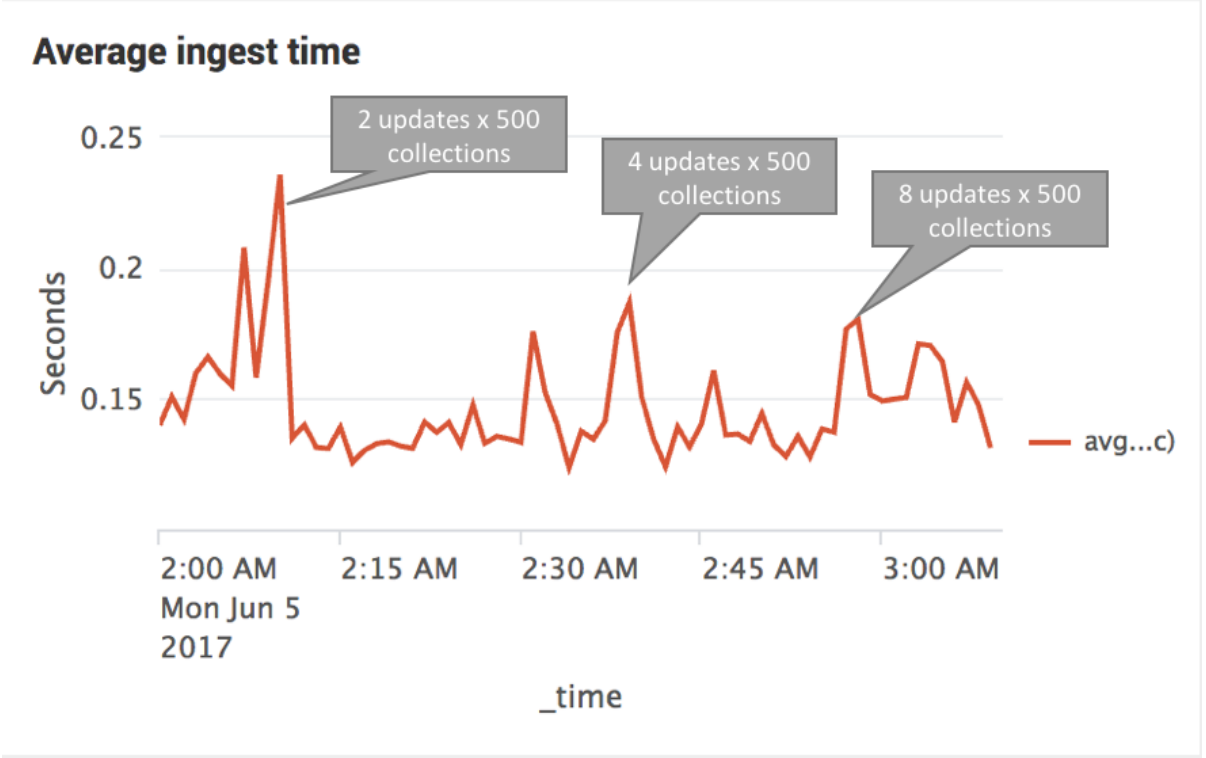
Ingest rate



Average ingest time







XML Update

During the translation of native XML to UMM, varying amounts of data are lost depending on the native XML type. To minimize or eliminate data loss, we explored updating native XML directly for bulk update. This presented many technical challenges and we were unable to come up with a way to do this that was efficient and maintainable, but came up with several approaches that we can further explore if we have to support native XML updates.

One thing that was discussed was having more robust functionality supported in UMM only, since UMM is vastly easier to update. If we go down a native XML update path, we could offer more limited bulk update functionality.

Technical Challenges/Considerations

1. Want to come up with an efficient, robust way to make these changes. Don't want to essentially reproduce custom translation code for each format making it difficult and costly to add new fields to be bulk updated and test.
2. Add to existing and clear all and replace are update types. If the field does not exist in the original XML, how do we know where to add it? For example, if we're adding to existing science keywords, but have a dif record that does not have any science keywords, how do we know where in the XML to add the science keywords? We would probably have to implement something that looks at the schema for each format.
3. The current XML libs being used in the CMR do not write to ISO, hence why we have custom XML gen code for when we do translations. The XML gen code does not have a parser, just a generator and even if it did, the vectors would not be easy to work with for updates. The XML libraries strip part of the ISO namespaces and don't handle nilReason well. We'd have to come up with a solution for this, possibly use straight java XML libs.

Approaches

These approaches are concepts that have been discussed with pros and cons.

Update XML in place - brute force approach

Pros:

- Do not lose data

Cons:

- Can make generic to a point, but will result in a lot of code, a lot of maintenance. Could try to leverage translation code with some refactoring.
- Will take a lot of work to implement new fields
- Problem of add to existing/clear and replace if field does not exist in original XML
- Issues writing iso xml

Make changes in UMM and copy back to XML

Convert the file to UMM, make the changes, convert back to native XML and copy just the update field back to the original XML using update functionality in XML libs.

Pros:

- Do not lose nearly as much data as an XML roundtrip
- A bit easier to make generic and implement, less code
- Need to do this anyway for umm-json

Cons:

- Lose some data - i.e. uuids in science keywords
- Prototyped to be generic(ish) with dif, dif10, echo10 science keywords. Even can get update working in ISO with some extra code. Still adding code for each update field and format probably.
- Problem of add to existing/clear and replace if field does not exist in original XML
- Issues writing iso xml

User sends up changes in XML

To make native XML updates, we'd have a separate API where the user could send update and xpath data for a specific format.

Pros:

- Less work updating XML
- Do not need to maintain a mapping of paths
- Lose no data

Cons:

- User would only be able to update one type of concept in one bulk update
- Need to prototype what user would send up and how it would work, how would find and replace/remove work?

- Problem of add to existing/clear and replace if field does not exist in original XML
- Issues writing to iso

UMM Roundtrip

This is pretty much what we're doing, but instead of the full roundtrip we are saving in umm-json.

Pros:

- By far easiest to implement and maintain
- Will need to do this anyway for umm-json collections
- Only approach that easily solves the problem of adding a field that does not exist in original XML

Cons:

- Max data loss of all approaches - does this provide enough value to end users?

Implement "Extended UMM"

The idea behind "extended UMM" is that it is a clojure map full of additional mappings that is only used for round-tripping. The end user would never see this in UMM. Without implementing UMM mappings, we'd take all of the fields we can't lose in each format, grab them from the XML, and during the roundtrip put them back in to the XML in the right place. This would all go in the current translation code, so we'd be using the vector structure and know the right place to insert the data.

Pros:

- Solves the problem of add to existing/clear and replace if field does not exist in original XML
- Solves the problem of not being able to write to ISO since it will fit in with our custom library
- Minimal data loss
- Easier to implement than UMM-C mappings
- No relying on UMM-C mappings

Cons:

- Some data loss - i.e. uuids in Science Keywords (or could implement with some work in extended umm)
- Will have to implement each missing field, adding to our translation code. Hard to test.
- Data loss until we implement those fields
- ISO will take a while to catch up - a lot missing right now and ISO is nested and complicated making the extended metadata implementation difficult.